**Universität Leipzig**
**Fakultät für Mathematik und Informatik**
**Institut für Informatik**

# Bachelorarbeit

Generierung von SPARQL Anfragen

Leipzig, 26. August 2011

vorgelegt von
Didier, Cherix
Studiengang Informatik

Betreuender Hochschullehrer:

Prof. Dr. Gerhard Brewka
Fakultät für Mathematik und Informatik
Intelligente Systeme

# Inhaltsverzeichnis

# 1 Einleitung

Semantic Web ist eine der größten aktuellen Herausforderungen in der Informatik. Um Daten einem semantischen Wert zuzuweisen, werden Ontologien benutzt. Ontologien definieren und verwalten Konzepte. Letztere beschreiben Objekte, haben Eigenschaften und was hier bedeutender ist, Relationen zueinander. Diese Konzepte und Relationen werden mit Hilfe einer Spezifikation (OWL zum Beispiel) charakterisiert. Diesen Konzepten werden Instanzen zugeordnet. Das heisst, dass beispielweise mit dem Konzept „Physiker“ die Instanz „Albert Einstein“ verbunden wird. Um zu erfahren, was „Albert Einstein“ mit der Stadt Berlin verbindet, gibt es Anfragesprachen, die bekannteste ist SPARQL.

Ohne Vorkenntnisse der Struktur einer Ontologie, ist es nicht möglich, präzise Anfragen zu erstellen. Die einzige Möglichkeit herauszufinden was zwei Instanzen verbindet, ist die Nutzung einer SPARQL-Anfrage mit Platzhaltern, also eine Anfrage auf Instanzebene durchzuführen. Es ist viel Aufwand nötig, um eine Anfrage auf Instanzebene zu lösen ohne vorher zu wissen, wie diese Instanzen miteinander verknüpft sein können. Um eine solche Anfrage Lösen zu können, müssen alle Relationen, die die erste Instanz betreffen, verfolgt werden, und von den so erreichten Instanzen diesen Vorgang weiterführen, bis die richtige Instanz gefunden wird. Die Instanzebene, auch A-Box genannt, ist die Ebene der tatsächlichen Elemente. Sie enthält zum Beispiel: „Berlin ist die Hauptstadt von Deutschland“ . Als erstes muss hierbei „Berlin“ als Instanz des Richtigen Konzept erkannt werden. In diesem Fall müsste also „Berlin“ als eine Instanz des Konzept „Stadt“ erkannt werden. Die Rückführung zu einem bestimmten Konzept wird in dieser Arbeit als gelöst betrachtet.

Das Ziel ist es, die Frage beantworten zu können, durch welche Instanzen $A$ und $B$ verbunden sind. Nimmt man das Beispiel was „Leipzig“ und „Dresden„verbindet, so ergibt sich der „Freistaat Sachsen“ . Wie vorhin bereits dargestellt ist es aber sehr aufwändig, die Instanz „Leipzig“ zu nehmen und zu prüfen, welche Relationen sie hat und zu welchen anderen Instanzen sie führt, und dies wiederum weiterzuführen bis die gefundene Instanz „Dresden“ ist.

In einem Graph gibt es Wege, diese Wege sind Verbindungen entlang der Kanten zwischen zwei Knoten. Diese Arbeit soll untersuchen, wie die Konzepte der Instanzen miteinander verbunden sind, um dann gezielt überprüfen zu können, welche Instanzen auf den gefundenen Wegen liegen. „Leipzig“ und „Dresden“ gehören beide zum Konzept „Stadt“ . Das Konzept „Stadt“ hat eine Relation, die zu dem Konzept „Bundesland“ führt. Wenn dieser Weg auf die Konzeptebene, T-Box genannt, gefunden worden ist, ist es möglich, mit SPARQL eine gezielte

Anfrage auf die A-Box zu stellen. Also zu fragen, welches „Bundesland“ verbindet „Leipzig“ und „Dresden“ .

Diese wird in den sich anschließenden Kapiteln folgendermaßen weitergeführt: Als erstes werden im Kapitel 2 die theorethischen Grundlagen kurz erklärt. Im Kapitel 3 werden die verwandten Arbeiten vorgestellt, anschließend wird im Kapitel 4 die Erstellung des Graphen der T-Box erklärt. Im Kapitel 5 wird auf die benutzte Suchtechnik eingegangen, im folgenden Kapitel 6 die Konvertierung eines gefundenen Weges in SPARQL erklärt. Im Kapitel 7 wird der gesamte Algorithmus kurz erklärt und seine Korrektheit nachgewiesen. Im Kapitel 8 wird der Algorithmus bezüglich seiner Vollständigkeit, Korrektheit und Performanz evaluiert, anschließend werden im Kapitel 9 die Ergebnisse diskutiert und im Kapitel 10 die möglichen folgenden Schritte vorgestellt.

# 2 Grundlagen

Dieser Artikel Arbeit nutzt einige Konzepte. Um Missverständnisse zu vermeiden werden hier die notwendigen Definitionen aufgeführt.

## 2.1 T-Box und A-Box

Eine Wissensdatenbank besteht aus zwei Komponenten : A- und T-Box. Letztere definiert die Terminologie, das heisst, den Wortschatz, der benutzt wird. Die A-Box enthält Aussagen über Individuen mit Hilfe des in der T-Box definierten Wortschatzes [BCM+03].

Die T-Box enthält also die Konzepte und die möglichen Relationen zwischen diesen, während die A-Box die Instanzen mit den tatsächlich auftretenden Relationen beinhaltet. In dieser Arbeit werden Wissensdatenbank und Ontologie als Synonyme verwendet.

## 2.2 RDF

Der *Resource Description Framework* (RDF) [KC06] ist eine vom W3C herausgegebene Spezifikation, um Aussagen über Resourcen zu beschreiben. RDF wird verwendet, um Resource zu identifizieren und Wissen über diesen zu speichern. Aussagen in RDF werden in der Form von Subjekt-Prädikat-Objekt Tripeln gespeichert. Es existieren mehrere Syntaxen, um die Aussagen zu beschreiben, zum Beispiel XML [BM04] oder Turtle [BBL08]. RDF-Schema (RDFS) [BG04]

ist eine Sprache, um RDF Wortschatz zu erzeugen.

## 2.3 SPARQL

**S**PARQL **P**rotocol **A**nd **R**DF **Q**uery **L**anguage (SPARQL) [PS+06] ist der Standard vom W3C, um RDF-Wissensdatenbanken abzufragen. Er basiert auf *Tripel-Pattern*. Ein Tripel-Pattern ist ein Tripel, von denen jeden Teil eine Variable sein kann [Leh10]. Mehrere *Trippel-Patterns* bilden zusammen einen *Graph-Pattern*.

## 2.4 OWL

Die **Web Ontology Language** [MvH+04] kurz **OWL** ist eine vom W3C definierte Spezifikationssprache. Es wird benutzt, um die Konzepte einer Ontologie und deren Relationen beschreiben zu können. OWL basiert auf RDF. OWL geht weiter als RDF und RDF-Schema [BG04]

## 2.5 Ungerichteter Graph

Ein ungerichteter Graph ist ein Paar $G = (V, E)$ mit $V \neq \emptyset$. $V$ ist die Menge der Knoten und $E$ die der Kanten. $E$ ist wie folgt definiert: $E \subseteq \mathcal{P}_2(V)$. $E$ ist eine Teilmenge der zweielementigen Potenzmenge von $V$ [OW96].

## 2.6 Gerichteter Graph

Ein Graph $G = (V, E)$ wird als gerichtet bezeichnet, wenn $E$ wie folgt definiert ist: $E \subseteq V \times V$, $E$ ist eine Teilmenge des kartesichen Produkts von $V$ mit $V$ [OW96].

## 2.7 Weg

Ein Kantenzug ist eine Folge $A_0, ..., A_n$ von Knoten, so dass $(A_i, A_{i+1}) \in E$ für $i = 0, ..., n-1$. Ein Kantenzug heißt Weg wenn alle Kanten verschieden sind. Ein Kantenzug ist gerichtet, wenn in einem gerichteten Graph $G$ kein Richtungwechsel entlang des Zuges vorkommt [OW96]. In dieser Arbeit sind alle Wege ungerichtet.

## 2.8 Vollständigkeit

Ein Algorithmus heisst vollständig, wenn er terminiert und eine Lösung liefert, falls eine solche existiert ([**?** ] Seite 75).

## 2.9 Korrektheit

Ein Algorithmus wird als korrekt bezeichnet, wenn er für jede Eingabe die richtige Lösung ausgibt [**?** ].

## 2.10 Grundlegendes Suchproblem

Diese Arbeit soll ermöglichen alle Instanzen, die auf die ungerichtete Wege zwischen zwei Instanzen liegen zu finden, damit in weiterführende Arbeiten, die Wegsuche weiter eingegrenzt werden kann. Es muss also von einem Konzept ausgehend alle Relation, entlang oder gegen deren Richtung, verfolgt werden können.

# 3 Verwandte Arbeiten

Unbekannte Ontologien abzufragen zu wollen, ist ein bekanntes Problem. Einige Techniken wurden schon untersucht, um dieses Problem zu vereinfachen. Darunter sind Ansätze über die A-Box, die A- und T-Box und die neue Spezifikation von SPARQL.

## 3.1 A-Box Ansätze

Der Relation-Finder [HHL$^+$09] nutzt nur die A-Box, um Objekte, die zwei Instanzen verbinden, zu finden. Dafür wird eine SPARQL-Anfrage benutzt, die alle Relationen von beiden Instanzen abfragt. Die Anfrage ist auf eine Länge von 20 begrenzt, somit können maximal 20 Relationen zwischen beiden Instanzen stehen. Außerdem ist die Anzahl an Umdrehungen innerhalb der Anfrage begrenzt. Hierbei bedeutet eine Umdrehung, dass $A$ über die Relation $r$ zu $B$ und $C$ über die Relation $r'$ zu $B$ gelangt. Dazu können Relationen oder Objekte explizit ignoriert werden.

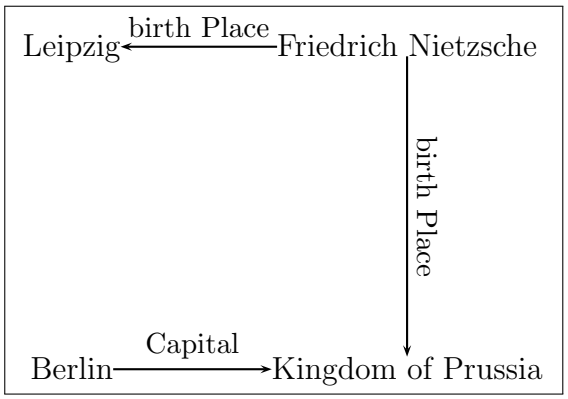



Abbildung 3.1: Um diese Verbindung zu finden, sind zwei Rich-

Der Nachteil dieses Ansatzes ist, dass es aufgrund der reinen A-Box-Suche mittels SPARQL sehr

langsam ist. Die Begrenzung der Anfrage auf eine Umdrehung kann die Ergebnisse einschränken. Beispielsweise findet der Relation-Finder einen wie in Abbildung 3.1 auf der vorherigen Seite gezeigten Weg nicht. Wenn „Nietzsche“ über einen anderen Weg, der maximal zwischen „Berlin“ und „Leipzig“ über eine Umdrehung verbunden ist, gefunden werden kann, ist es nicht weiter tragisch. Es kann aber nicht davon ausgegangen werden, dass es für alle über mehr als eine Umdrehung verbundenen Instanzen der Fall ist.

## 3.2 A- und T-Box Ansätze

Der Semantic wiki search Ansatz [HHMT09] nutzt sowohl die A-Box als auch die T-Box. Da es sich um einen Keyword-Search-Ansatz handelt, ist die A-Box mit keywords angereichert. Um eine performante Suche durchführen zu können, wird die angereicherte A-Box auf eine mit Keywords angereicherte T-Box reduziert. Um dies zu erreichen werden nur bedeutende Kanten mitgenommen. Die Auswahl dieser Kanten erfolgt über ein Ranking. Für jede Relation in der T-Box wird geschaut wie oft sie tatsächlich in der A-Box auftritt. Besonders werden Kanten, die von „owl:Thing“ zu „owl:Thing“ also von allem zu allem definiert sind, auf die tatsächlich auftretenden Domains und Ranges in der A-Box reduziert.

Es werden verschiedene Kostenfunktionen genutzt [TWRC09], unter anderem die Länge der gefundenen Verbindungen, und eine Ranking-Funktion die wie folgt definiert ist:

$$c(v) = 1 - \frac{|v_{agg}|}{|V|} \; c(e) = 1 - \frac{|e_{agg}|}{|E|}$$

wobei $v_{agg}$ der Anzahl an Knoten in der A-Box, die zu einem Knoten in der T-Box gehören und $|V|$ der Anzahl an Knoten in der T-Box entsprichen, $e_{agg}$ und $|E|$ sind für die Kanten analog.

Dieser Ansatz verbietet Zyklen, was bei vielen Ontologien zu Problemen führen kann (siehe Abschnitt 5.1 auf Seite 15)

## 3.3 SPARQL 1.1

SPARQL 1.1 [HS10] kann einen Teil des Problems lösen, da es über Property Paths verfügt, die die Anfrage kürzer und einfacher zu gestalten erlaubt. Ein Property Path ist ein möglicher Weg zwischen zwei Knoten innerhalb eines Graphen. Ein Property Path der Länge 1 gleicht einem Tripel. Zyklen sind möglich und ein Path der Länge 0 verbindet einen Knoten mit sich selbst. Diese Property Paths vereinfachen zwar die Syntax der Anfrage, werden aber auf der Ebene der A-Box gelöst. Das heisst, es werden auf Instanzebene alle möglichen Relationen abgearbeitet, die anstelle der Property Paths auftreten können. Der Aufwand ist also je nach Instanz und Länge der Property Path sehr hoch.

Außerdem sind einige Optimierungen eingeführt worden, die die Geschwindigkeit erhöhen können. SPARQL 1.1 ist momentan noch in der Entwicklungphase.

# 4 Grapherstellung

[NM09] Als erste Aufgabe muss ein Graph der T-Box erstellt werden. Dafür wurden sowohl die theorethischen Grundlagen der OWL-Spezifikation [MvH+04] als auch mehrere Ontologien betrachtet und deren Sonderfälle berücksichtigt.

## 4.1 Theorethische Grundlagen

Der Graph der T-Box bildet diese direkt ab. Die Knoten des Graphen entsprechen die Klassen, „owl:Class“. Die Relationen, „owl:ObjectProperty“, werden als Kanten eingetragen. Dazu wird die Hierarchie, „rdfs:subClassOf“, auch durch Kanten gespiegelt.

### 4.1.1 Komplexe Klassen

In RDF beziehungsweise OWL sind einige komplexe Klassen definiert. Diese können nicht ohne weiteres direkt in den Graph eingetragen werden. Folgende Abschnitte beschreiben, wie diese komplexen Klassen behandelt worden sind.

**„IntersectionOf“** „IntersectionOf“ ist äquivalent zu einer Konjunktion [siehe MvH+04]. Jedoch gibt es zwei verschiedene Fälle: zum Eine, wenn „IntersectionOf“ nur Objekte, und zum Anderen wenn „IntersectionOf“ ein Objekt und eine Relation betrifft.

Wenn „IntersectionOf“ sich nur auf Objekte bezieht, wird die Relation für alle in der „IntersectionOf“ auftretenden Objekte erzeugt.

```
<owl:ObjectProperty rdf:about="R">
  <rdfs:domain>
    <owl:Class rdf:about="A" />
  </rdfs:domain>
  <rdfs:range>
      <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="A" />
        <owl:Class rdf:about="C" />
      </owl:intersectionOf>
  </rdfs:range>
</owl:ObjectProperty>
```

Also in diesem Fall werden Kanten von A nach B und A nach C im Graph eingetragen.

Wenn allerdings „IntersectionOf“ sich auf ein Objekt und eine Relation bezieht, sind die Instanzen betroffen, die diese Relation erfüllen. Zum Beispiel ist das Objekt „Employee“ in der LUBM Ontologie [GPH05] definiert als die „IntersectionOf“ „Person“ zur Relation „worksFor“. Das Ziel des Graphen ist es mögliche Wege zu finden, um eine SPARQL-Anfrage zu konstruieren. Die semantische (A-BOX) Überprüfung ist nicht relevant und „Employee“ kann einfach als eine Unterklasse von „Person“ modelliert werden. Dazu wird die Relation „worksFor“ von „Person“ auf „Employee“ übertragen (siehe Abbildung 4.1 auf der nächsten Seite).

In der LUBM [GPH05] ist die Klasse „Student“ wie folgt in OWL angegeben:

```
<owl:Class rdf:ID="Student">
  <rdfs:label>student</rdfs:label>
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Person" />
     <owl:Restriction>
       <owl:onProperty rdf:resource="#takesCourse" />
         <owl:someValuesFrom>
           <owl:Class rdf:about="#Course" />
         </owl:someValuesFrom>
```

```
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

Die Abbildung 6.1 auf Seite 18 zeigt wie eine solche komplexe Klasse behandelt wird.

**„UnionOf“** „UnionOf“ ist äquivalent zu einer Disjunktion. Eine Relation, die auf eine komplexe Klasse vom Typ „UnionOf“ zeigt oder von einer stammt, ist nur gültig für die Instanzen der A-Box, die zu allen Konzepten der „UnionOf“ gehören. Der Graph repräsentiert nur die T-Box, folglich kann die nächste Relation irgendeine, die eine der Konzepte von dem „UnionOf“ betreffen, sein. Am einfachsten, um die Anzahl an Knoten und Kanten gering zu halten, ist es, eine Kante für jedes Konzept innerhalb von „UnionOf“ zu erzeugen.

**„SubpropertyOf“** Laut der Definition von „SubpropertyOf“ [MvH+04] ist $P_1$ „SubpropertyOf“ $P_2$, wenn die Menge aller Tupel der durch $P_1$ verbundenen Instanzen eine Teilmenge der Menge aller durch $P_2$ verbundenen Instanzen ist. Für die T-Box gibt es also zwei mögliche Fälle: Erstens, dass Ranges und Domains von $P_1$ gleich die von $P_2$ sind. In diesem Fall kann $P_1$ mit denselben Ranges und Domains wie $P_2$ in den Graph eingetragen oder ausgelassen werden, wenn bei dem SPARQL-Endpoint die Inference eingeschaltet ist. Mit der Inference ist gemeint, dass der Endpoint über einen Reasoner verfügt und dieser fähig ist von einer oberen Relation auf die „subPropertyOf“ dieser zurück zu führen. Der zweite Fall ist, wenn sich Domains oder Ranges zu denen von $P_2$ unterscheiden. In diesem Fall kann $P_1$ wie eine normale Relation betrachtet oder ausgelassen werden, wenn bei dem SPARQL-Endpoint die Inference eingeschaltet ist.

## 4.2 Betrachtete Ontologien

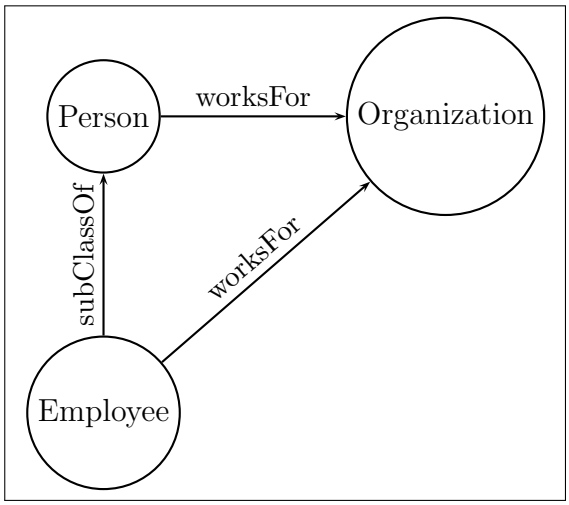



Abbildung 4.1: „IntersectionOf“ on property

Um die Erstellung des Graphen so allgemeingültig wie möglich zu gestalten, wurden mehrere Ontologien betrachtet. Bei der Betrachtung der im Anschluss vorgestellten Ontologien wurden einige notwendigen Anpassungen an den Graphen vorgenommen.

### 4.2.1 Ontologien

**LUBM** Die LUBM [GPH05] modelliert eine Universität und wurde als erste Ontologie benutzt. Für diese Ontologie war die richtige Behandlung als „IntersectionOf on Property“ nötig.

**„Pizza Ontologie“** Die „Pizza Ontologie“ [DHS+07] ist eine Beispielontologie in OWL. Besonders zu beachten ist hierbei die „IntersectionOf on class“. Da die Pizza Ontologie keine Instanzen besitzt, wird sie nicht für die Evaluierung betrachtet.

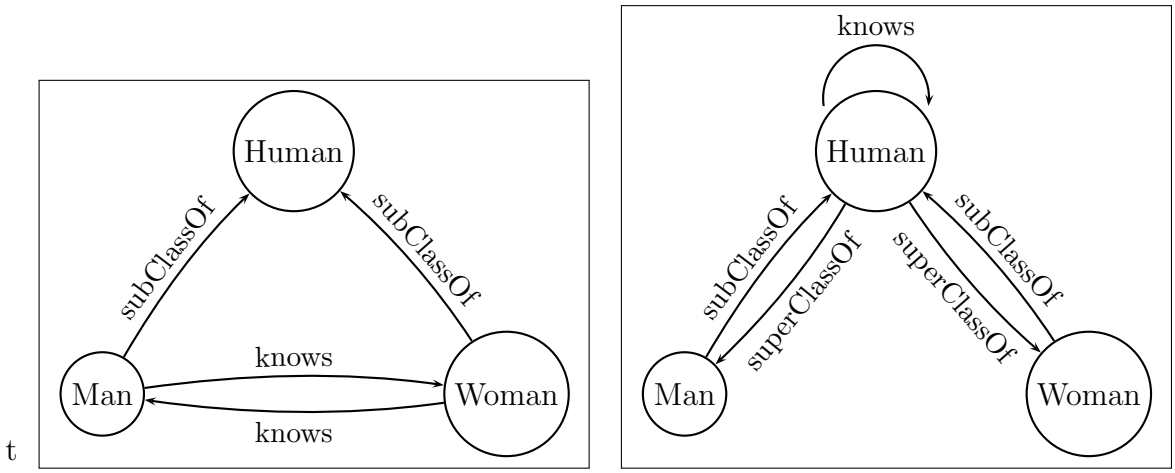
**„Neue Testament Ontologie“** Die „Neue Testament Ontologie“ [Boi11] betrachtet die Namen der Personen, die im neuen Testament vorkommen. Für diese Ontologie war es notwendig, Zyklen zulassen zu können.

**DBpedia** Die „DBpedia Ontologie“ ist eine explizite Ontologie, die Wissen aus Wikipedia abbildet [BCAK07]. Daraus folgt, dass sie weder komplexe Klassen, wie „IntersectionOf“, noch iterative Properties enthält. Die sich daraus ergebende Herausforderung ist die Größe der T-Box. Die Anzahl an möglichen Wegen zwischen zwei Objekten der T-Box ist sehr hoch und es ist nicht möglich, alle in einer annehmbaren Zeit zu finden. Die Anzahl an gesuchten Wegen muss also begrenzt werden und es muss möglich sein, zu beeinflussen, welche Wege zuerst gefunden werden.

### 4.2.2 Abgeleitete Fälle

Aus den theoretischen Grundlagen und den beobachteten Ontologien ergab sich, dass zwei Hauptgraphtypen für diese Arbeit notwendig wurden: gerichtet oder ungerichtet.

**Gerichteter Graph** Ein gerichteter Graph hat den Vorteil, dass die Relationen der Ontologie direkt übernommen werden können. Die Hierarchie der verschiedenen Objekte ist auch trivial zu repräsentieren.




t

(a) Einfache Modellierung der „knows“ Relation (b) Modellierung der „knows“ Relation wie in der NTN [Boi11] beschrieben

**Abbildung 4.2:** Gerichteter Graph für eine einfache Ontologie

Wie Abbildung 4.2a zeigt, ist eine einfache Ontologie problemlos mit einem gerichteten Graph darzustellen. Die Ontologie in Abbildung 4.2a ist jedoch etwas unglücklich gewählt, da sie nur zulässt, dass ein „Man“ eine „Woman“ kennt und umgekehrt. Jedoch nicht, dass ein „Man“ einen „Man“ kennt. Um dies zu ermöglichen, ist es die einfachste Lösung, die Relation „knows“ auf die Ebene „Human“ zu übertragen. Eine solche Relation ist aber mit der bisherigen Grapherzeugung insbesondere mit Hilfe einer einfachen Tiefensuche leider nicht zu finden. Die Suche beginnt fast immer an einer der untersten Knoten (ein Blatt des Hierarchiebaums) und endet auch an einem solchen. Um eine Relation zwischen „Man“ und „Woman“, wie in Abbildung 4.2b gezeigt, zu finden, wird die Kante von „Man“ zu „Human“ verfolgt, anschließend die Kante „knows“ von „Human“ zu „Human“ und würde ohne die Einführung von „superClassOf“ Kanten in „Human“ feststecken. Diese Kanten sind notwendig, um von einem Punkt $A$ bis zu einem niedrigeren Punkt $B$ zu gelangen, wenn es Relationen gibt, die in höheren Stufen der Hierarchie eingetragen sind. Leider ermöglicht ein solcher Graph nur gerichtete Wege. Es ist somit nicht möglich, einen Weg zu finden, dessen Richtung wechselt. Um zum Beispiel eine Vater-Sohn Verbindung, wie in Abbildung 4.3 auf der nächsten Seite gezeigt zu finden, müsste der Graph ungerichtet sein.

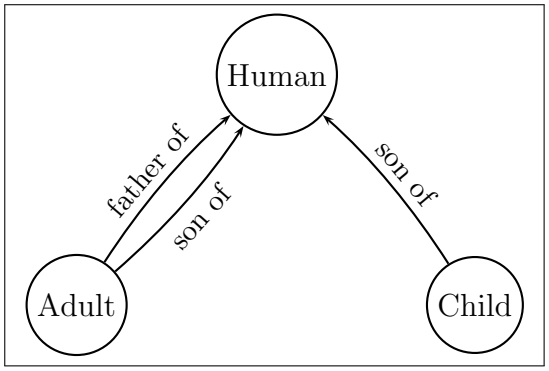



Abbildung 4.3: Vater-Sohn Modellierung

**Ungerichteter Graph** Ein ungerichteter Graph löst das Problem des Richtungswechsel. Neue Probleme ergeben jedoch die Kanten mit Inversen. Durch die Inversen entstehen zwischen zwei verschiedenen Knoten zwei ungerichtete, anstelle von zwei gerichteten Kanten. Dies kann später bei der Suche den Aufwand extrem erhöhen und lässt zwei semantisch äquivalente Pfade, im Gegensatz zu zwei verschiedene entstehen. Eine Lösung wäre eine der beiden Kanten zu eliminieren, um die Wege und später die SPARQL Anfragen verständlicher zu gestalten. Eine andere wäre es, die Kanten mit Inverse als gerichtet einzutragen; so ist immer die richtige Relation gewählt und die Wege sind einfacher in SPARQL zu konvertieren und nachzuvollziehen.

**Kantengewichtung** In sehr großen Ontologien, wie zum Beispiel DBPedia, kann es nützlich sein, Kanten (auf Ontolgieebene Relationen) zu priorisieren. Die Priorisierung der Kanten kann als Sortierung derselben und somit als Reihenfolge in der sie betrachtet werden verstanden werden oder als Gewichtung im allgemeinen Sinne. Beide Ansätze sind relativ gleich. Es muss nur entschieden werden, ob eine Kante mit einer niedrigeren Gewichtung wichtiger oder unwichtiger ist. Für kürzeste Wege-Algorithmen sind die Gewichtungen als Kosten zu verstehen und somit hat beispielweise eine „wichtigere“ Kante eine niedrigere Gewichtung.

# 5 Suche

Um die verschiedenen Wege in einem Graph zu finden braucht man einen Suchalgorithmus. In den meisten Fällen wird eine Antwort nicht ausreichen. Die T-Box enthält mehrere Wege zwischen zwei Konzepte, die A-Box muss aber nicht alle diese Wege für jede der Instanzen enthalten. Deswegen kann eine Antwort nicht ausreichen und es eignen sich in erster Linie Algorithmen, die eine Traversierung des Graphen gewährleisten.

Einige spezielle Eigenschaften sind zusätzlich noch erforderlich. Die Suchmethode darf Wege, die nur über die Hierarchie gehen bzw. die ohne Verwendung einer Relation die Hierarchie hoch- und heruntergehen (siehe Abbildung 5.1 auf Seite 16), nicht nutzen.

## 5.1 Zyklen

Da einige Relationen dasselbe Konzept als Domain und als Range besitzen müssen Zyklen zugelassen werden. Außerdem ist es nicht ungewöhnlich einen Weg mit dem selben Start- und Endpunkt zu suchen, wie zum Beispiel in der DBpedia Ontologie einen Weg von `http://dbpedia.org/ontology/person` zu `http://dbpedia.org/ontology/person`. Die verwendeten Algorithmen lassen Zyklen der Länge 1 zu, um die Anzahl an Möglichkeiten zu verringern und um vor allem sinnlose Wege zu unterbinden. Das Erlauben von Zyklen erfordert, dass ein Knoten zweifach besucht werden kann. Um weitere sinnlose Wege zu vermeiden muss das Verfolgen derselben Kante zweimal hintereinander verboten werden.

Leider gibt es einige Fälle, bei denen mehrmaliges Ablaufen eines Zyklus erforderlich sein kann. Wie in Abbildung 4.2b auf Seite 13 gezeigt, ist es üblich, dass die „knows“ Relation dasselbe Objekt als Domain und Range besitzt. Das Zulassen von Zyklen der Länge 1 ermöglicht eine Verbindung der Form $A$ *kennt* $B$ zu finden. Leider ist es aber auf dieser Weise nicht möglich eine Verbindung zwischen $A$ und $C$ über den Weg $A$ *kennt* $B$ und $B$ *kennt* $C$ zu finden. Die beste Lösung wäre wahrscheinlich eine Sonderregel für solche Kanten einzuführen. Das bedeutet, man müsste für diese Kanten einen Wert definieren, der die Häufigkeit, wie oft eine Kante hintereinander abgelaufen werden kann, und die Anzahl an erlaubten Besuch des als Domain und Range dienenden Knotens festlegt. Dieses sollte am besten per Hand gemacht werden, da nur die Bedeutung der Kante hilft, den richtigen Wert zu finden.

## 5.2 Tiefensuche

Die Tiefensuche ist auf den ersten Blick nicht die richtige Wahl, da sie nicht vollständig ist. Das heisst, es besteht das Risiko, dass kein Weg gefunden wird, obwohl einer existiert. Wenn zum Beispiel ein Zyklus existiert, könnte dieses unendlich viele Male hintereinander abgelaufen werden und so die Verfolgung eines anderes Weg, der die Lösung enthält, blockieren. Ontologien sind aber so aufgebaut, dass es fast immer möglich ist, zwei verschiedene Konzepte miteinander über sehr lange Wege zu verbinden. Meistens haben diese Wege aber wenig Sinn und es gibt keine Resultate auf die entsprechende SPARQL-Anfrage. Zum Beispiel ist es möglich über eine Relation von „Thing“ zu „Thing“ jedes Konzept miteinander zu verbinden. Je länger der Weg wird, desto unwahrscheinlicher ist es, dass es tatsächlich Instanzen gibt, die zu diesem Weg gehören. Deswegen ist es sinnvoll, die Länge der Wege zu begrenzen. Diese Begrenzung hätte den Effekt, dass eine Tiefensuche für Wege. die kürzer als die von der Begrenzung festgelegte

Länge sind, vollständig wäre.

Zusätzlich ist die Bildung der Wege einfacher, da die Tiefensuche lediglich einen von diesen folgt. Bei einer Breitensuche werden nur Teilstücke benutzt, die am Ende wieder zusammengesetzt werden müssten. Die Priorisierung der Kanten funktioniert hiermit auch problemlos. Die hochpriorisierten Kanten werden als erste gewählt und verfolgt. Wenn die Kanten eines vorher gewählten Weges mit der höchsten Priorität versehen werden, findet die modifizierte Tiefensuche diesen Weg als erstes.

Die Behandlung der priorisierten Kanten erfolgt so, dass die mit dem besten Wert immer als erstes bearbeitet werden. Dies hat den Nachteil, dass ein Weg dessen mittlere Kanten sehr hoch und dessen Anfangskanten sehr niedrig priorisiert sind, später als ein Weg, dessen Anfangskante eine höhere Priorisierung haben, gefunden wird, obwohl die gesamte Priorisierung niedriger ist.

## 5.3 Iterative Tiefensuche

Die iterative Tiefensuche hat den Vorteil, dass sie zuerst immer die kürzesten Verbindungen findet. Im Gegensatz dazu ist sie auch weniger durch die Kantenpriorisierung beeinflussbar, da ein Weg $w$ der Länge $l$, bei dem alle Kanten die höchste Priorisierung haben, erst gefunden werden kann, wenn alle Wege $w'$ mit Länge $l' \leq l$ gefunden worden sind. $w$ wird als erster Weg unter den Wegen der Länge $l$ gefunden. Die iterative Tiefensuche hat allerdings den Nachteil, dass sie bei einem Graph mit Hierarchie, einen Weg, der über sehr viele „subClassOf“ Kanten geht, erst später findet, obwohl eine tatsächliche Länge (=Summe der eigentlichen Relationen) existiert, die viel kürzer ist.

## 5.4 „Kürzeste-Wege-Algorithmen“

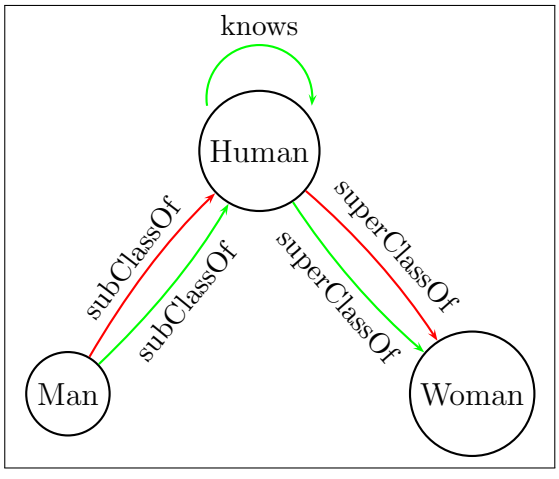



Abbildung 5.1: Der rote Weg ist nicht zugelassen. Der grüne Weg ist zugelassen

Mit Hilfe einer als Kosten verstandenen Priorisierung der Kanten, könnten Algorithmen für kürzeste Wege benutzt werden. Leider ergeben diese Algorithmen, zum Beispiel Azevedo-Algorithmus [**?** ], meistens nur einen Weg als Lösung. Algorithmen, die die $k$ kürzesten Wege als Lösung ergeben, berechnen ihn, und nach einem Umbau des Graphen, der den kürzesten Weg verbietet, den zweitkürzesten und

„FullProfessor“ „subClassOf“ „Professor“,
„Professor“ „subClassOf“ „Faculty“,
„Faculty“ „teacherOf“ „Course“.

so weiter. Der Aufwand ist also hoch und kann sich nur lohnen, wenn erstmal die richtige Kostenfunktion entwickelt worden ist. Aus diesem Grund wurden solche Algorithmen nicht verwendet.

# 6 Konvertierung in SPARQL

Die Konvertierung des Weges in eine SPARQL-Anfrage ist relativ einfach und für alle Graphtypen gleich. Der Weg kann einfach verfolgt werden. Dabei werden die „subClassOf“ und „superClassOf“ Kanten ignoriert. Bei den anderen Kanten wird der Name der Range mit einer sich anschließenden Zufallszahl als Variable genutzt. Das einzige was beachtet werden muss, sind Richtungswechsel. Um dies zu gewährleisten, wird der Range der vorherigen Kanten mit der Domain der aktuellen verglichen. Sind beide Werte gleich, muss nichts weiteres beachtet werden und die letzte Variable wird genutzt. Sind beide Werte ungleich muss eine neue Variable eingeführt und die vorherige als Range benutzt werden. Dazu wird im nächsten Schritt die Domain zur Range der letzten Kante.

Die daraus entstehenden SPARQL-Anfragen sind einfach, das heisst sie sind ohne Filter-, Union-, Optionalanweisungen, und somit einfache konjuktive Anfragen.

```
SELECT * WHERE {
?place <http://dbpedia.org/ontology/place>
<http://dbpedia.org/resource/Leipzig>
?place ... }
```

Aus den in Abbildung 6.1 auf der nächsten Seite vorgestellten Graphen lassen sich zwei Wege von „FullProfessor“ zu „Course“ ermitteln. Der erste Weg sieht so aus:

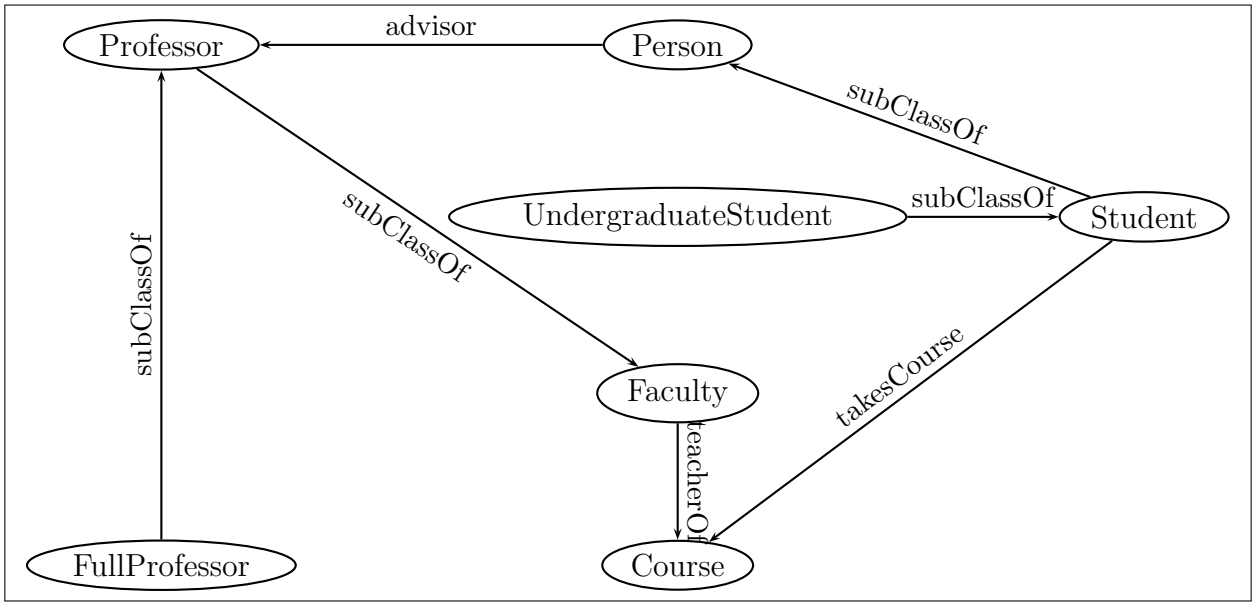



Abbildung 6.1: Verbindung in der LUBM

| | | |
|---|---|---|
| „FullProfessor“ | „subClassOf“ | „Professor“, |
| „Person“ | „advisor“ | „Professor“, |
| „Person“ | „superClassOf“ | „Student“, |
| „Student“ | „takesCourse“ | „Course“. |

Der zweite so:

Die durch die Konvertierung erzeugten SPARQL-Anfragen sind wie folgt aufgebaut:

```
SELECT * WHERE {
<Course1> ?s <FullProfessor0> }
LIMIT 100
```

```
SELECT * WHERE {
?Person2 <advisor> <FullProfessor0> .
?Person2 <takesCourse> <Course1> . }
LIMIT 100
```

# 7 Algorithmus

Der gesamte Algorithmus ist wiefolgt aufgebaut. Der erste Schritt ist die Erstellung des Graphen, dannach folgt die Suche, und anschließend die Konvertierung in SPARQL.

**Algorithm 1:** Gesamt Algorithmus

**input** : Startinstanz SI, Zielinstanz ZI
**input** : Starttyp ST, Zieltyp ZT
**input** : Ontology O
**output**: SPARQL Query

```
1 G ←Creategraph(O);
2 W ← Search(G, ST, ZT);
3 foreach w ∈ W do
4 |  Convert(w);
5 end
```

## 7.1 Validierung

In den folgenden Abschnitten werden die 3 Regeln des Algorithmus nachgewiesen.

### 7.1.1 `Creategraph()`

Diese Regel erstellt den Graphen. Wie im Kapitel 4.1 beschrieben, spiegeln die Knoten die Konzepte der T-Box und die Kanten die möglichen Relationen wider. Um den Graph zu erstellen, wird die Ontologie traversiert und bei einem Besuch von „owl:class“ ein neuer Knoten im Graph eingefügt, falls dieser noch nicht existiert. Dazu werden „subClassOf“ und „supperClassOf“ Kanten zu bzw. von den in der Ontologie als „rdfs:subclassOf“ eingetragenen Objekte geführt. Falls keine „rdfs:subclassOf“ existieren, laufen diese Kanten zu „Thing“ hin beziehungsweise von „Thing“ weg. Bei Besuch einer „owl:ObjectProperty“ wird eine Kante hinzugefügt, falls sie noch nicht existiert.

Die Traversierung gewährleistet, dass kein Konzept und keine Relation ausgelassen wird. Bei komplexe Klassen wird wie im Kapitel 4.1 beschrieben vorgegangen. Der Graph entspricht also einer Darstellung der T-Box und ist dementsprechend gleich definiert.

### 7.1.2 `Search()`

Sowohl die begrenzte wie die iterative Tiefensuche sind vollständig. Dies bedeutet, dass ein Weg gefunden wird, wenn einer existiert. Da die Suchalgorithmen so modifiziert sind, dass sie nicht abbrechen, wenn sie einen ersten Weg gefunden haben, sondern weiter laufen, finden sie alle möglichen Wege. Die weitere Suche auf einem Graphen $G$ nach dem Fund von einem Weg $W$ ist äquivalent zu einer Suche im Graph $G'$, wobei $G'$ gleich $G$ ohne den Weg $W$ ist. Da die Suche vollständig ist wird also ein neuer Weg $W'$, der ungleich $W$ ist, gefunden, bis es keine neue Wege mehr gibt. Es werden also alle möglichen Wege gefunden.

### 7.1.3 `Convert()`

Bei der Konvertierung eines Weges werden alle Kanten nacheinander bearbeitet. Bei dieser Bearbeitung treten folgende Fälle auf :

- Die Kante ist vom Typ „subClassOf“:
  Die Funktion `Convert()` ignoriert eine solche Kante und merkt sich nur deren Range für die weitere Bearbeitung.

  Die Relation „rdfs:subClassOf“ ist eine Relation auf der T-Box Ebene. Ihre Domains und Ranges sind Konzepte, also Mengen von Instanzen. Dies bedeutet, dass eine Instanz eine solche Relation nicht haben kann, aber dass die Konzepte, zu denen sie gehört, in einer solchen Relation stehen können. Daraus folgt, dass eine Anfrage `SELECT * WHERE {s rdfs:subClassOf ?o}`, wenn `s` eine Instanz entspricht keine Ergebnisse liefern kann.

- Die Kante ist vom Typ „superClassOf“:
  Da „superClassOf“-Kanten eigentlich eine Inverse von einer „rdfs:subClassOf“-Relation darstellen, werden sie wie die korrespondierenden Kanten behandelt und einfach ignoriert.

- Die Domain der Kante ist gleich dem Range der vorherigen Kante:
  In diesem Fall ist kein Richtungswechsel vorhanden und die Kante entspricht einem SPARQL-Tripel.

- Der Range der Kante ist gleich der Domain der vorherigen Kante:
  In diesem Fall ist ein Richtungwechsel vorhanden. Da die SPARQL-Tripeln und die Relationen einer Ontologie gerichtet sind, und somit eine Relation von $A$ nach $B$ definiert ist und muss in SPARQL die Abfrage über einen Tripel $\{A\ rel\ B\}$ erfolgen. Eine Anfrage

der Form $\{B\ rel\ A\}$ würde keine Antwort geben. Aus diesem Grund muss die Kante umgedreht und der Range als erstes eingetragen werden.

# 8 Evaluierung

In dem folgenden Abschnitt wird der Algorithmus auf Vollständigkeit, Korrektheit und Performanz evaluiert. Dafür wurden verschiedene Tests entworfen und durchgeführt. Anschließend werden die Ergebnisse kurz vorgestellt.

## 8.1 Vollständigkeit

Wie in Kapitel 2 beschrieben, ist ein Algorithmus vollständig, wenn er eine Lösung findet, falls eine existiert. Um diese Eigenschaft testen zu können, müssen Eingaben gesucht werden, von denen die Existenz einer Lösung bekannt ist.

### 8.1.1 Tests

Mit den oben vorgestellten Ontologien wurden aufbauähnliche Tests durchgeführt. Diese Tests sind wie folgt entwickelt worden: Zwischen den Instanzen der Ontologie wurden zwei Instanzen ausgewählt, die eindeutig direkt oder über andere Instanzen verbunden sind, ausgewählt. Dabei sind beispielweise folgende Tests entstanden:

**V1** In der Neue-Testament-Ontologie [Boi11] wird eine Verbindung der Länge 1 gesucht. Startknoten ist `<http://semanticbible.org/ns/2006/NTNames#Priscilla>` zu `<http://semanticbible.org/ns/2006/NTNames#Urbanus>`. Die Verbindung ist direkt über die Relation `<http://semanticbible.org/ns/2006/NTNames#knows>` möglich.

**V2** Eine etwas kompliziertere Suche ist eine mit dem Startpunkt `<http://semanticbible.org/ns/2006/NTNames#Priscilla>` und dem Endpunkt `<http://semanticbible.org/ns/2006/NTNames#Aristobulus>`. Dabei geht die kürzeste Verbindung über `<Priscilla>` wohnt (`<residentPlace>`) in `<Rome>` und `<Aristobulus>` auch.

**V3** In der LUBM mit nur einer Universität wird eine Verbindung zwischen `<http://www.Department1.University0.edu/FullProfessor0>` und `<http://www.Department1.University0.edu/Course1>` gesucht. Als Ergebnis dieser Anfrage werden `<UndergraduateStudent151>`

| Graphtype | Lösung gefunden |
|---|---|
| Gerichtet | Ja |
| Semigerichtet | Ja |
| Ungerichtet | Ja |

Tabelle 8.1: Ergebnisse von V1

| Graphtype | Lösung gefunden |
|---|---|
| Gerichtet | Ja |
| Semigerichtet | Ja |
| Ungerichtet | Ja |

Tabelle 8.2: Ergebnisse von V2

und `<teacherOf>` erwartet. Da es mehrere Lösungen gibt und das Ziel dieses Algorithmuses darin besteht alle Instanzen zu finden, muss die Definition der Vollständigkeit erweitert werden: Ein Algorithmus ist vollständig wenn er alle existierenden Lösungen findet. Die Verbindung ist in der Abbildung 6.1 auf Seite 18 abgebildet.

### 8.1.2 Ergebnisse

| Graphtype | Undergraduate Student | teacherOf |
|---|---|---|
| Gerichtet | Nein | Ja |
| Semigerichtet | Ja | Ja |
| Ungerichtet | Ja | Ja |

Tabelle 8.3: Ergebnisse V3

## 8.2 Korrektheit

Unter Korrekteit wird die Eigenschaft die richtige Lösung zu finden verstanden. Die Korrektheit kann wie die Vollständigket getestet werden.

### 8.2.1 Tests

Die benutzten Tests sind gleich die für die Vollständigkeit.

### 8.2.2 Ergebnisse

Die Ergebnisse entsprechen denen der Vollständigkeit, Das bedeutet die gefundenen Lösungen waren immer die richtigen.

## 8.3 Performanz

Für die Performanztests wurde die DBpedia verwendet. Die DBpedia wurde gewählt, weil diese Ontologie oft benutzt wird und keine künstlich erzeugte, womöglich vereinfachte Ontologie ist. Performanztests sind leider sehr schwer durchzuführen, da sich die Suchzeiten sehr verlängern können. Mit Verwendung des Virtuoso SPARQL-Endpunkt von DBpedia auf `http://dbpedia.org/sparql` ist es nicht möglich einen Referenzwert zu berechnen, da dies die zulässige Rechenzeit überschreiten würde. Selbst bei einer eigenen DBpedia-Instanz würden diese Timeouts vorkommen, da die RDFS-Graph Servern zu viel Zeit benötigen. Die von Virtuoso geschätzten Antwortzeiten erstrecken sich von 20 Stunden bis Tagen.

### 8.3.1 Tests

**P1** Als Startpunkt wurde `<http://dbpedia.org/resource/Leipzig>` und als Endpunkt `<http://dbpedia.org/resource/Napoleon_I_of_France>` gewählt. Die korrespondierten Konzepte sind respektiv `http://dbpedia.rog/ontology/City` und `http://dbpedia.org/ontology/Person`. Die Abbildung 8.1 auf der nächsten Seite zeigt wie das Konzept `<City>` zu `<Person>` verbunden ist. Die maximale Verbindungslänge beträgt 3. Es ergibt sich somit als reine SPARQL Query folgende Anfrage:

```
SELECT ?p ?o1 ?o2 ?o3 WHERE {
{<Leipzig> ?p ?o1} UNION {?o ?p <Leipzig>}
```

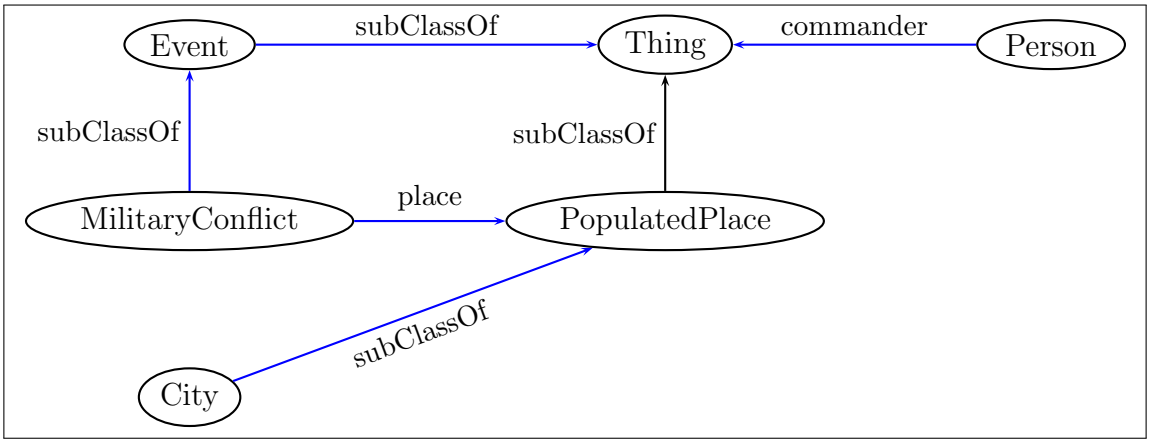



Abbildung 8.1: In blau Weg von Leipzig zu Napoleon

```
OPTIONAL {{?o1 ?p1 ?o2} UNION {?o2 ?p1 ?o1}}

OPTIONAL {{?o2 ?p2 ?o3} UNION {?o3 ?p2 ?o2}}

OPTIONAL {{?o3 ?p3 ?o4} UNION {?o4 ?p3 ?o3}}

FILTER ( ?o1=<Napoleon_I_of_France> || ?o2=<Napoleon_I_of_France> ||
    ?o3=<Napoleon_I_of_France> ). }
LIMIT 100
```

---

**P2** Eine einfachere SPARQL-Anfrage von `<Airline>` zu `<Airline>` mit einer maximalen Länge 2 ist bereits zu aufwändig für die DBpedia. Dagegen ergeben sich folgende Werte bei einer Maximallänge von 3 und einer maximalen Anzahl von Anfragen bzw. Wegen von 1000, mit Hilfe der iterativen Tiefensuche.

### 8.3.2 Ergebnisse

Die Tabelle 8.4 zeigt, wieviel Zeit nötig ist, um den Graph zu erstellen, um die Suche von

| Aufgabe [ms] | Graphtype | | |
|---|---|---|---|
| | Gerichtet | Semi-gerichtet | Ungerichtet |
| Graph erstellen | 845 | 4439 | 856 |
| Suche | 1527 | 4058 | 3210 |
| Antwort | keine | 410 | 486 |
| Gesamt | 2356 | 8907 | 3552 |

Tabelle 8.4: Benötigte Zeit je Teilaufgabe in Millisekunden

maximal 1000 Wegen durchzuführen und um eine Antwort vom SPARQL-Endpoint zu bekommen. Der gerichtete Graph findet keine Antwort, da er keinen Richtungswechsel zulässt. Der Gesamtzeit in diesem Graph entspricht der Zeit, bis alle gefundenen Wege mittels SPARQL angefragt worden sind. Bei den anderen Graphen ist die gesamte Zeit die, die gebraucht wurde, um eine Antwort zu bekommen. Die für diese Suche benötigten Graphen enthalten keine Relationen, die von „Thing“ zu „Thing“ laufen, um die Anzahl an Wegen zu reduzieren. Das führt zu dem Ergebnis, dass der Ansatz zwar korrekt ist, leider aber zeitlich noch nicht ganz ausgereift ist für eine Ontologie wie DBpedia. Außerdem ist dieser Weg früh in dem Graph zu finden. Das heisst, die Kanten, die diesen weg bilden, werden als erste ausgewählt und somit wird der ganze Weg sehr schnell gefunden. In diesem Fall ist die Kantenpriorisierung gleich der lexikographischen Ordnung. Daraus ergibt sich also auch, dass die Suche sehr viel länger dauern kann.

# 9 Diskussion

Die in dieser Arbeit genutzten Graphtypen verhalten sich etwas verschieden zu einander. Wie die Tabellen 8.1 und 8.2 zeigen sind alle 3 Graphtypen für einfache Verbindungen geeignet. Jedoch Tabelle 8.3 zeigt, dass der gerichtete Graph nicht alle Wege findet. Dies liegt an der gerichteten Eigenschaft des Graphen. Anders formuliert, dieser Graph ermöglicht lediglich gerichtete Wege zu finden, also Wege ohne Richtungswechsel. Die im Abschnitt 8.1.2 vorgestellten Tabellen sind nur eine Auswahl der durchgeführten Tests. Es wurden ähnlichen Test auf allen hier vorgestellten Ontologien durchgeführt und die Ergebnisse waren analog. Es konnten immer Wege, wenn welche existierten, gefunden werden, mit eine Ausnahme : den gerichteten Graph.

Die Tests mit der DBpedia zeigen, dass die Anzahl an möglicher Wege innerhalb der T-Box sehr groß werden können und über eine Priorisierung der Kanten nachgedacht werden sollte. Der im Abschnitt 8.3.1 vorgestellte Test kann dafür leicht verändert werden. Die Priorisierung der Kanten entlang des Weges, der, um die erwartete Lösung zu finden, genutzt werden soll, werden per Hand auf den höchsten Wert gesetzt. So wird sehr schnell eine korrekte Lösung gefunden.

Die in Tabelle 8.4 vorgestellten Ergebnisse zeigen, dass wenn die Priorisierung der Kanten günstig ist, eine Lösung relativ schnell gefunden werden kann. In diesem Fall war, wie im Abschnitt 8.3.2 beschrieben, die Priorisierung gleich der lexikographische Ordnung. Die Performanz hängt also von zwei Hauptfaktoren ab: Entweder ist die Ontologie relativ klein, das heisst die T-Box enthält wenig verschiedene Konzepte und Relationen, oder wenn dies nicht der Fall ist, da die DBpedia ungefähr 300 Konzepte enthält, muss die Priorisierung der Kanten richtig gewählt werden. Eine einfache Priorisierung mit Hilfe der Anzahl von Ergebnissen für einen Weg hat mehrere Nachteile. Zum einem müssen alle Wege berechnet werden, was eine sehr lange Vorarbeit verusachen kann, zum anderen beeinflussen sich die Wege gegenseitig, vor allem wenn für alle Konzepte priorisiert werden soll.

Die verschiedenen Graphtypen zeigen, dass beim Semigerichteten Graph die Grapherstellung viel länger dauert, die zusätzlichen Kanten aber kaum Vorteile im Vergleich zum ungerichteten Graph bringen. Dies liegt aber zum Teil an der DBpedia-Ontologie, die keine Inverse benutzt. Die Überprüfung auf Inverse verlangsamt nur die Grapherstellung ohne Vorteil zu bringen.

Somit kann festgestellt werden, dass die Analyse des T-Box Graphen für eine kleine Ontologie ausreichen kann, jedoch nicht bei einer grösseren T-Box. Mögliche weitere Lösungen werden im nächsten Kapitel vorgestellt.

# 10 Ausblick

Diese Arbeit hat gezeigt, dass sich zwei große Themen für die weitere Forschung ergeben. Zum ersten wie bereits beschrieben, ist die Kantenpriorisierung ein sehr bedeutendes Thema. Dazu ist die Reduktion beziehungsweise Optimierung der T-Box nötig. Diese Optimierung könnte sowohl manuell, also über eine neue Definition der Ontologie oder dynamisch, mit Hilfe der tatsächlichen Instanzen, erfolgen.

Der Nachteil am dynamischen Ansatz ist, dass bei einer Ontologie wie DBpedia so viele Instanzen existieren, dass es kaum möglich sein wird, Relationen enger zu definieren. Wie in

[TWRC09] beschrieben, ist eine Art Pageranking möglich. Dieses ist aber nur auf die LUBM-Ontologie getestet worden. Somit könnte es problematisch werden bei einer Ontologie wie DBpedia, die extrem viele Instanzen besitzt.

Eine bessere Möglichkeit wäre zum Beispiel den Graph in Regionen unterteilen zu können, also in Teilgraphen, die Konzepte enthalten, die sehr einander annähern.

## 10.1 Gewichtung der Kanten

Die hier benutzte manuelle Methode zur Gewichtung der Kanten soll mit einer statistischen verglichen werden. Dieser Vergleich soll aber unbedingt über die DBpedia erfolgen, da sie eine sehr große T-Box besitzt. Wie in [TWRC09] beschrieben ist eine Art Pageranking möglich.

## 10.2 Preprocessing

Eine weitere Möglichkeit, die Performanz zu erhöhen, ist es, die Wege vorzuberechenen, und zum beispielweise Wichtungen anhand der Egrenissanzahl oder anhand einer Evaluation der Antworten. Diese Evalauation sollte prüfen, ob die Ergebnisse tatsächlich dem Gesuchten entsprechen.

## 10.3 Disambiguierung

Insgesamt wäre es möglich, die Anfrage innerhalb des Prozesses zu verfeinern: sei es direkt über Rückfrage des Endbenutzer oder über eine statistische Auswertung des Kontextes.

## 10.4 Indexierung des Graphen

Eine Möglichkeit würde sich auch aus der Indexierung des Graphen ergeben. Dazu könnte man Mengen von Klassen betrachten. Zum Beispiel die Menge der Klasse, die als Domain der Relation x gelten und dazu die Menge der Klassen, die als Range der Klasse x gelten. So könnte man im nächsten Schritt die Relationen überprüfen, die als Teil- der Rangemenge gelten. Also wäre zu untersuchen, ob dieser Ansatz weniger Schritte während der Suche benötigt, und ob es sich speichertechnisch besser implementieren lässt als mit einem Graph.

# Literatur


[BBL08] D. Beckett and T. Berners-Lee. Turtle-terse rdf triple language. *W3C Submission, Jan*, 2008.

[BCAK07] C. Bizer, R. Cyganiak, S. Auer, and G. Kobilarov. DBpedia—querying wikipedia like a database. In *Developers track presentation at the 16th international conference on World Wide Web, WWW*, pages 8–12, 2007.

[BCM+03] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[BG04] Dan Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. *W3C recommendation* `http://www.w3.org/TR/rdf-schema/`, 2004.

[BM04] D. Beckett and B. McBride. RDF/XML syntax specification (revised). *W3C recommendation* `http://www.w3.org/TR/rdf-syntax-grammar/`, 10, 2004.

[Boi11] Sean Boisen. New testament names: a semantic knowledge base. *Retrieved from* `http://www.semanticbible.com/ntn/ntn-overview.html`, April 2011.

[DHS+07] N. Drummond, M. Horridge, R. Stevens, C. Wroe, and S. Sampaio. Pizza ontology v1. 5, 2007.

[GPH05] Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2-3):158–182, 2005.

[HHL+09] Philipp Heim, Sebastian Hellmann, Jens Lehmann, Steffen Lohmann, and Timo Stegemann. RelFinder: Revealing Relationships in RDF Knowledge Bases. In *Proceedings of the 4th International Conference on Semantic and Digital Media Technologies (SAMT 2009)*, pages 182–187, Berlin/Heidelberg, 2009. Springer.

[HHMT09] Peter Haase, Daniel Herzig, Mark Musen, and Thanh Tran. Semantic wiki search. In *ESWC 2009*, LNCS, pages 445–460, Berlin, 2009. Springer.

[HS10] S. Harris and A. Seaborne. Sparql 1.1 query language. *Working draft, W3C* ***http://www.w3.org/TR/sparql11-query/***, 2010.

[KC06] G. Klyne and J.J. Carroll. Resource description framework (rdf): Concepts and abstract syntax. *W3C recommendation* ***http://www.w3.org/TR/rdf-concepts/***, 2006.

[Leh10] J. Lehmann. *Learning OWL Class Expressions*. PhD thesis, Universität Leipzig Fakultät für Mathemathik und Informatik, `http://nbn-resolving.de/urn:nbn:de:bsz:15-qucosa-38351`, 2010.

[MvH+04] D.L: McGuiness, F. van Harmelen, et al. OWL Web Ontology Language overview. *W3c recommendation* ***http://www.w3.org/TR/owl-features***, 10:2004–03, 2004.

[NM09] Natalya Noy and Mark Musen. Traversing ontologies to extract views. In Heiner Stuckenschmidt, Christine Parent, and Stefano Spaccapietra, editors, *Modular Ontologies*, volume 5445 of *Lecture Notes in Computer Science*, pages 245–260. Springer Berlin / Heidelberg, 2009.

[OW96] Thomas Ottmann and Peter Widmayer. *Algorithmen und Datenstrukturen (German Edition)*. Spektrum Akademischer Verlag, 1996.

[PS+06] E. Prud'Hommeaux, A. Seaborne, et al. Sparql query language for rdf. *W3C working draft*, 4, 2006.

[TWRC09] T. Tran, H. Wang, S. Rudolph, and P. Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. In *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*, pages 405–416. IEEE, 2009.